DESIGN, IMPLEMENTATION, AND PERFORMANCE OF MPI ON PORTALS 3.0

Ron Brightwell[1]
Rolf Riesen[1]
Arthur B. Maccabe[2]

Summary

This paper describes an implementation of the Message Passing Interface (MPI) on the Portals 3.0 data movement layer. Portals 3.0 provides low-level building blocks that are flexible enough to support higher-level message passing layers, such as MPI, very efficiently. Portals 3.0 is also designed to allow for programmable network interface cards to offload message processing from the host processor, allowing for the ability to overlap computation and MPI communication. We describe the basic building blocks in Portals 3.0, show how they can be put together to implement MPI, and describe the protocols of our MPI implementation. We look at several key operations within the implementation and describe the effects that a Portals 3.0 implementation has on scalability and performance. We also present preliminary performance results from our implementation for Myrinet.

Key words: Portals, MPI, Myrinet, Message Passing, Cplant

1 Introduction

The emergence of cluster computing as a viable platform for high performance computing has been realized due to significant performance increases in commodity computing and networking hardware. In particular, relatively inexpensive programmable network interface cards (NICs), such as Myrinet (Boden et al., 1995), that are capable of delivering gigabit-per-second speeds, have allowed for much research into low-level message passing protocols and message passing interfaces (von Eicken et al., 1992, 1995; Ishikawa et al., 1996; Pakin et al., 1997; Myricom, Inc., 1997; Compaq et al., 1997; Prylli and Tourancheau, 1998). Most of this research has been focused on delivering latency and bandwidth performance as close as possible to the limitations of the hardware.

In several aspects, the research on clusters of personal computers (PCs) with gigabit networking hardware is addressing many of the same problems that proprietary distributed-memory message passing parallel machines of the early 1990s faced. Despite the differences in hardware architecture between custom-built parallel machines and today's PC cluster, many of the issues with respect to delivering network performance to parallel applications are similar.

The Portals (Brightwell et al., 1999, 2002) data movement interface (Portals 3.0) is an evolution of networking technology initially developed for large-scale, distributed memory, massively parallel systems. Portals began as a key component of our lightweight compute node operating systems (Maccabe et al., 1994; Shuler et al., 1995), and has evolved into a functional interface that can be implemented efficiently for different operating systems and networking hardware. In particular, Portals provides the necessary building blocks for higher-level protocols to be implemented on programmable or intelligent network interfaces without providing mechanisms that are specific to each higher-level protocol. This paper describes how these building blocks and their associated semantics can be combined to support the protocols needed for a scalable, high performance implementation of the Message Passing Interface (MPI) Standard (MPI Forum 1994). Portals is the basis for the Computational Plant (Cplant™) (Brightwell et al., 2000) cluster at Sandia National Laboratories, and the MPI implementation described in this paper has been used on our large-scale production machines for the last two years.

[1] CENTER FOR COMPUTATION, COMPUTERS, INFORMATION, AND MATHEMATICS, SANDIA NATIONAL LABORATORIES, U.S.A.

[2] DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF NEW MEXICO, U.S.A.

The International Journal of High Performance Computing Applications, Volume 17, No. 1, Spring 2003, pp. 7–19

The rest of this paper is organized as follows: In the next section, we give a brief overview of the Portals 3.0 API. In Section 3, we present the initial implementation of MPI on Portals 3.0. Section 4 describes a problem that we encountered with this implementation and Section 5 describes a second implementation of MPI that uses a new semantic added to Portals to overcome this limitation. We discuss the benefits of our implementation with respect to progress in Section 6. We present performance data in Section 7 and summarize the key points of this paper in Section 8. We conclude with a discussion of ongoing and future work in Section 9.

2 Portals 3.0

The Portals 3.0 API is composed of elementary building blocks that can be combined to implement a wide variety of higher-level data movement layers such as MPI. We have tried to define these building blocks and their operations so that they are flexible enough to support other layers in addition to MPI. For example, the Cplant™ parallel runtime system (Brightwell and Fisk, 2001) is built on top of Portals, and there is currently an effort underway to build a parallel file system on top of Portals (Braam et al., 2002). However, MPI was certainly our main focus. The following sections describe Portals objects and their associated functions.

The Portals library provides a process with access to a virtual network interface. Each network interface has an associated Portal table that contains at least 64 entries. The table is simply indexed from 0 to $n-1$, and the entries in the table normally correspond to a specific high-level protocol. Portal indexes are like port numbers in Unix. They provide a protocol switch to separate messages intended for different protocols.

Data movement is based on one-sided operations. Other processes can use a Portal index to read (get) or write (put) the memory associated with the remote Portal. Each data movement operation involves two processes, the **initiator** and the **target**. In a put operation, the initiator sends a put request containing the data to the target. The target translates the Portal addressing information using its local Portal structures. When the request has been processed, the target may send an acknowledgment. In a get operation, the initiator sends a get request to the target. The target translates the Portal addressing information using its local Portal structures and sends a reply with the requested data.

Typically, one-sided operations use a triple to address remote memory: <process id, buffer id, offset>. In addition, Portal addresses include a set of match bits. Figure 1 presents a graphical representation of the structures used to translate Portal addresses. The process id is used to route the message to the target node. The buffer id is used as an index into the Portal table, and this identifies a match list. The match bits (along with the id of the initiator) are used to select a match entry (ME) from the match list. The match entry identifies a list of memory descriptors (MDs). An MD identifies a logically contiguous region of user memory and optionally contains an event queue (EQ).

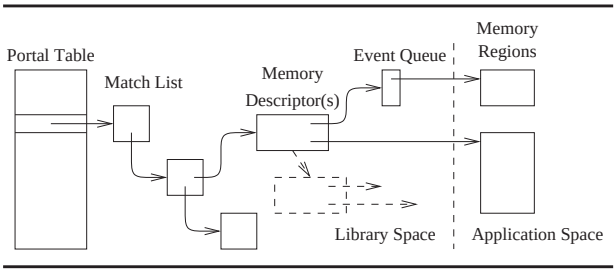

Fig. 1 Portal addressing structures.

A match entry (ME) provides message selection based on: the initiator, 64 match bits, and 64 ignore bits. The initiator can be "wildcarded" to allow matching with any process id. Match bits are 64-bit values that can be used for further selection. The match bits can be selectively "wildcarded" using the ignore bits.

The MEs in a match list are searched in sequential order. If an ME matches the request, the first memory descriptor associated with the ME will be tested for acceptance of the message. If the ME does not match the message or the MD rejects the message, the message continues to the next ME in the list. If there are no further MEs, the message is discarded. An ME also has the option of being unlinked from the list after it has been consumed. The following section will help to define what it means for an ME to be consumed.

Memory descriptors have a number of options that can be combined to increase their utility. They can be configured to respond to put operations, get operations, or both. Each MD has a threshold value that determines how many operations can occur before it becomes inactive. In addition, each MD has an offset value which can be managed locally or remotely. When it is managed locally, the offset is increased by the length of each message that is deposited into the MD. Consecutive messages will be placed in the user's memory one after the other. MDs can specify that an incoming message which is larger than the remaining space will be truncated or rejected.

By default, MDs generate acknowledgments to the process that initiated the successful operation. An incoming put operation can request that an acknowledgment be delivered to the originating process. An acknowledgment contains information about the result of the operation at the destination process. The decision to generate

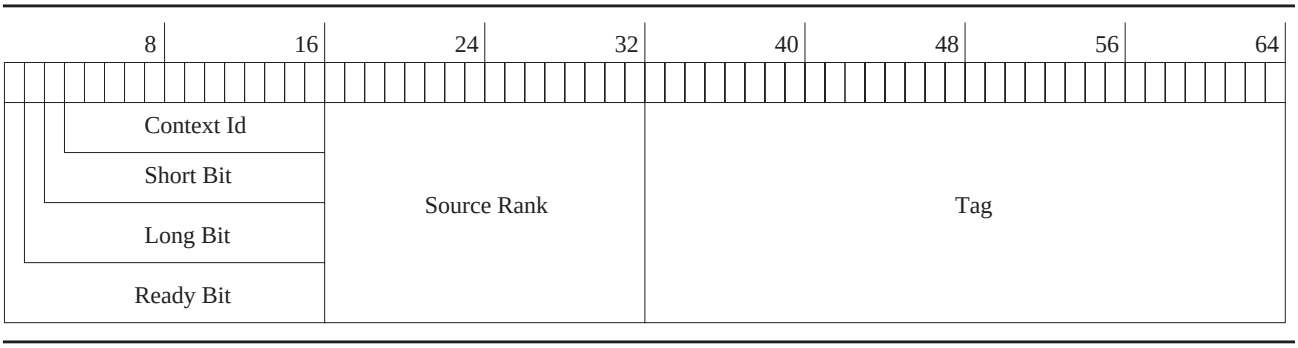

Fig. 2 Match bits.

an acknowledgment in response to an operation requires input from both the initiator and the target. The initiator must request an acknowledgment and the MD used in the operation must be configured to generate an acknowledgment.

MDs can be chained together to form a list. Moreover, each MD can be configured to be unlinked from the list when it is consumed (i.e., when its threshold becomes zero). When the last MD in the list is consumed and unlinked, the associated ME also becomes inactive. Each ME can also be configured to unlink when it becomes inactive.

MDs may have an associated event queue (EQ). EQs are used to record operations that have occurred on MDs. Multiple MDs can share a single EQ, but an MD may only have one EQ.

EQs are composed of individual events kept in a circular buffer in the application's address space. There are five types of events that represent the five operations that can occur on an MD:

Get Event Generated when an MD responds to a get request.

Put Event Generated when an MD accepts a put operation.

Sent Event Generated when it is safe to manipulate the memory region used in a put operation.

Reply Event Generated when the reply from a get operation is stored in an MD.

Ack Event Generated when a acknowledgment arrives from the target process.

In addition to the type of event, each event records the state of the MD at the time the event occurred.

3 Initial MPI Implementation

In this section, we describe our MPI implementation for Portals 3.0. This implementation is a port of MPICH (Gropp et al., 1996) version 1.2.0. It uses a two-level protocol, based on message size, to optimize for short message latency and optimize for bandwidth for large messages. In addition to message size, the different protocols are used to meet the semantics of the different MPI send modes.

3.1 MATCH BITS

Figure 2 shows how the 64 match bits are used. We use the match bits to encode the send protocol (3 bits), the MPI communicator (13 bits), the local rank (within the communicator) of the sending process (16 bits) and the MPI tag (32 bits). During `MPI_Init()`, we set up three Portal table entries. The *receive Portal* is used for receiving MPI messages. The *read Portal* is used for unexpected messages in the long message protocol. The *ack Portal* is used for receiving acknowledgments for synchronous mode sends. We also allocate space for handling unexpected messages, the implementation of which we will describe below. After these structures have been initialized, all of the processes in the job call a barrier operation to ensure that all have been initialized.

3.2 SHORT MESSAGE PROTOCOL

In our implementation, we use an eager protocol for short messages (standard and ready sends). The entire user buffer is sent immediately and "unexpected messages" (when there is no pre-posted receive) are buffered at the receiver.

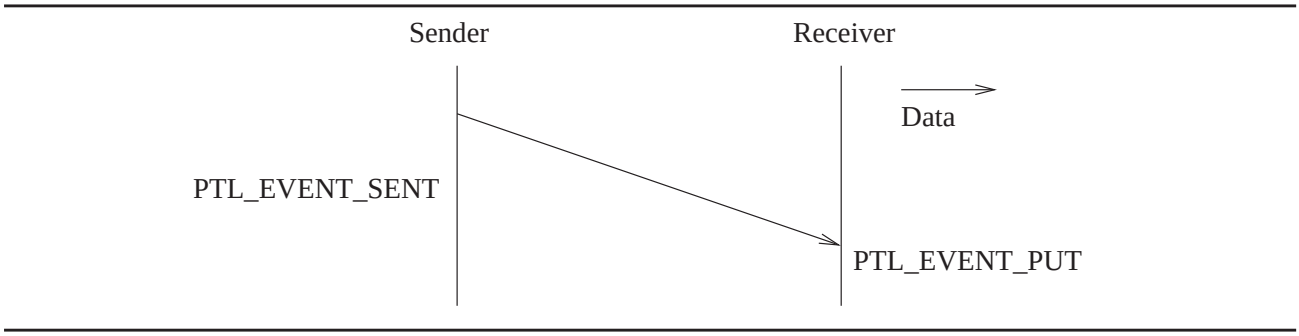

Fig. 3 Short standard send protocol.

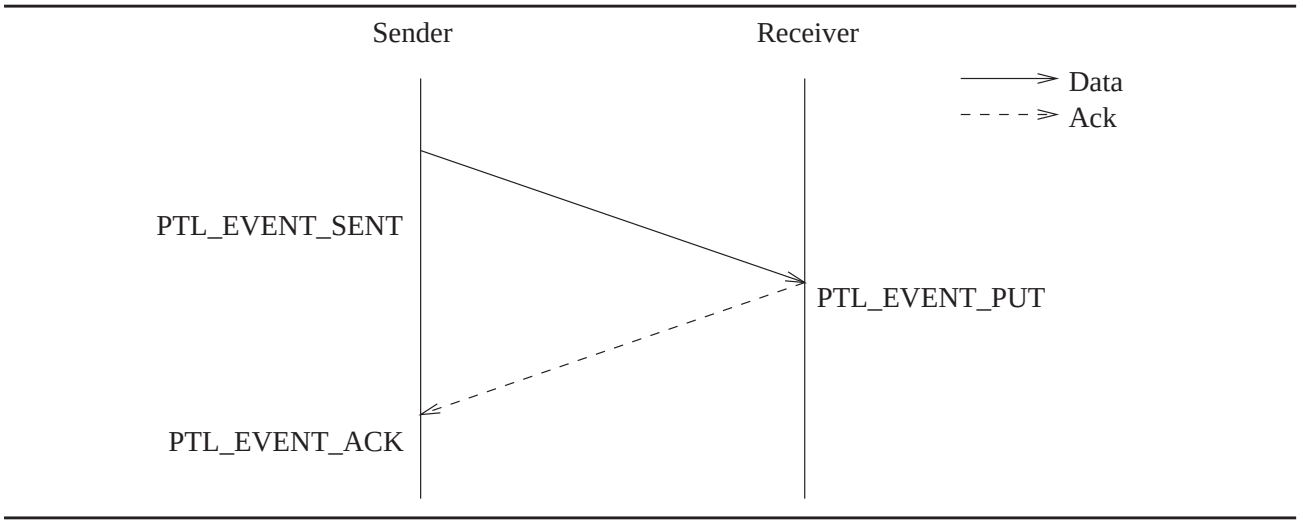

Fig. 4 Short synchronous send protocol (expected).

For standard sends, we allocate an MD to describe the user buffer and an EQ associated with the MD. We configure the MD to respond to put operations and set the threshold to one. We create an address structure that describes the destination and fill in the match bits based on the protocol and MPI information. Using the Portal put function, we send the MD to the *receive Portal* of the destination process, requesting that no acknowledgment be sent. This send is complete when an event indicating that the message has been sent appears in the EQ. Figure 3 presents a timing diagram for the short send.

For synchronous sends, we need to know when the message is matched with a receive. We start by allocating an MD and an EQ as described earlier. Next, we build an ME that uniquely matches this message. This ME is associated with the MD allocated in the previous step and attached to the local *ack Portal*. We configure the MD to respond to acknowledgments and put operations and set the threshold to two. When the Portal put operation is called, we request an acknowledgment, and include the match bits for the ME in 64 bits of out-of-band data, called the header data.

Completion of a short synchronous mode send can happen in one of two ways. If the matching receive has been posted, an acknowledgment is generated by the remote memory descriptor as illustrated in Figure 4. If the matching receive is not posted, the message is buffered until the matching receive is posted. At this point, the receiver will send an explicit acknowledgment message using the Portal put operation, as illustrated in Fig-

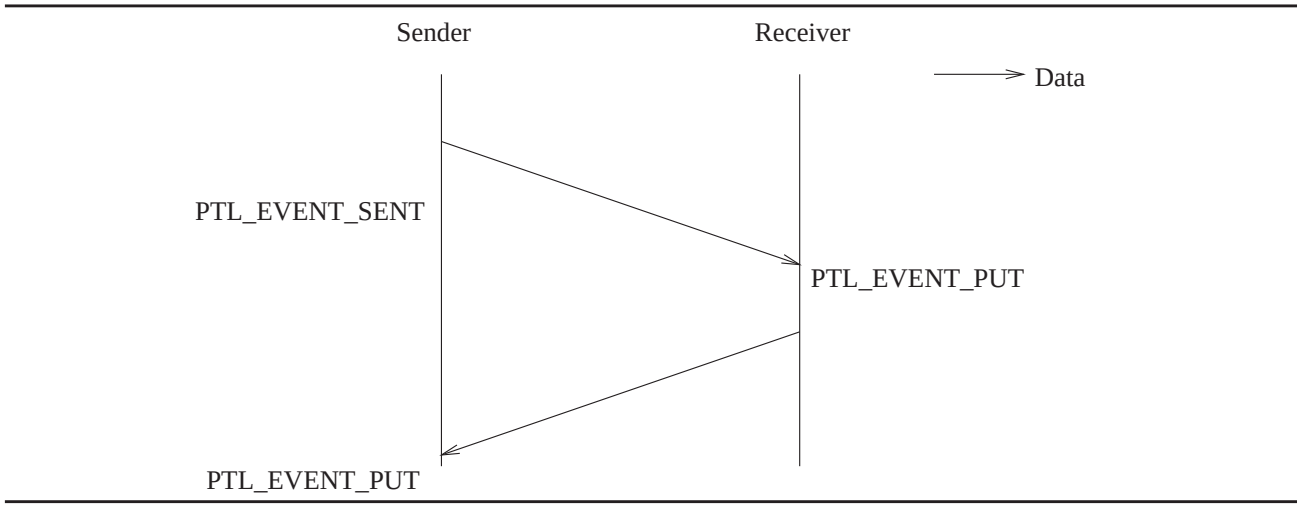

Fig. 5 Short synchronous send protocol (unexpected).

ure 5. A short synchronous mode send completes when the `PTL_EVENT_SENT` event has been recorded and either the `PTL_EVENT_ACK` event or the `PTL_EVENT_PUT` event has been recorded.

3.3 LONG MESSAGE PROTOCOL

Unlike most MPI implementations, we also use an eager protocol for long messages. That is, we send long messages assuming that the receiver has already posted a matching receive. Because the message could be discarded, we also make it possible for the receiver to get the message when it does post a matching receive.

We start by inserting an ME that uniquely describes this send on the *read Portal*. We create an MD that describes the user buffer. This MD is configured to respond to put operations, get operations, and ack operations. Since all three of these may occur, we set the MD's threshold to three. We then attach the MD to the ME. We set the protocol bits in the match bits for a long send and fill in the other match bits appropriately. We call the Portal put function, requesting an acknowledgment, and we include the match bits of the ME in the header data.

As with the short synchronous mode sends, the long send protocol can complete in one of two ways. First, if the message is expected at the receiver, an acknowledgment is returned. In this case, the event queue will contain a `PTL_EVENT_SENT` event and a `PTL_EVENT_ACK` event. After these two events have been recorded, the send is complete. This is illustrated in Figure 6.

If the message was not expected, an acknowledgment is returned to the sender indicating that the receiver accepted zero bytes. The sender must then wait for the receiver to request the data from the sender's read Portal. In this case, three events mark the completion of the send: a `PTL_EVENT_SENT` event, a `PTL_EVENT_ACK` event, and a `PTL_EVENT_GET` event. This is illustrated in Figure 7.

Since the completion of this send protocol is dependent on a matching user buffer being posted at the receiver, this protocol is the same for standard mode and synchronous mode sends. Moreover, the ready mode send for long messages is identical to a short standard mode send. Since the MPI semantics guarantee that a matching buffer is posted at the receiver, the sender need not wait for an acknowledgment or set up an entry on the read Portal.

3.4 POSTING RECEIVES

Posting a receive involves two lists: the posted receive list and the unexpected message list. The unexpected message list holds messages for which no matching receive has been posted. Before a receive can be added to the posted receive list, we must search the unexpected list. Figure 8 illustrates the match list structure we use to represent these two lists. This list starts with entries for the posted receives (no entries are shown), followed

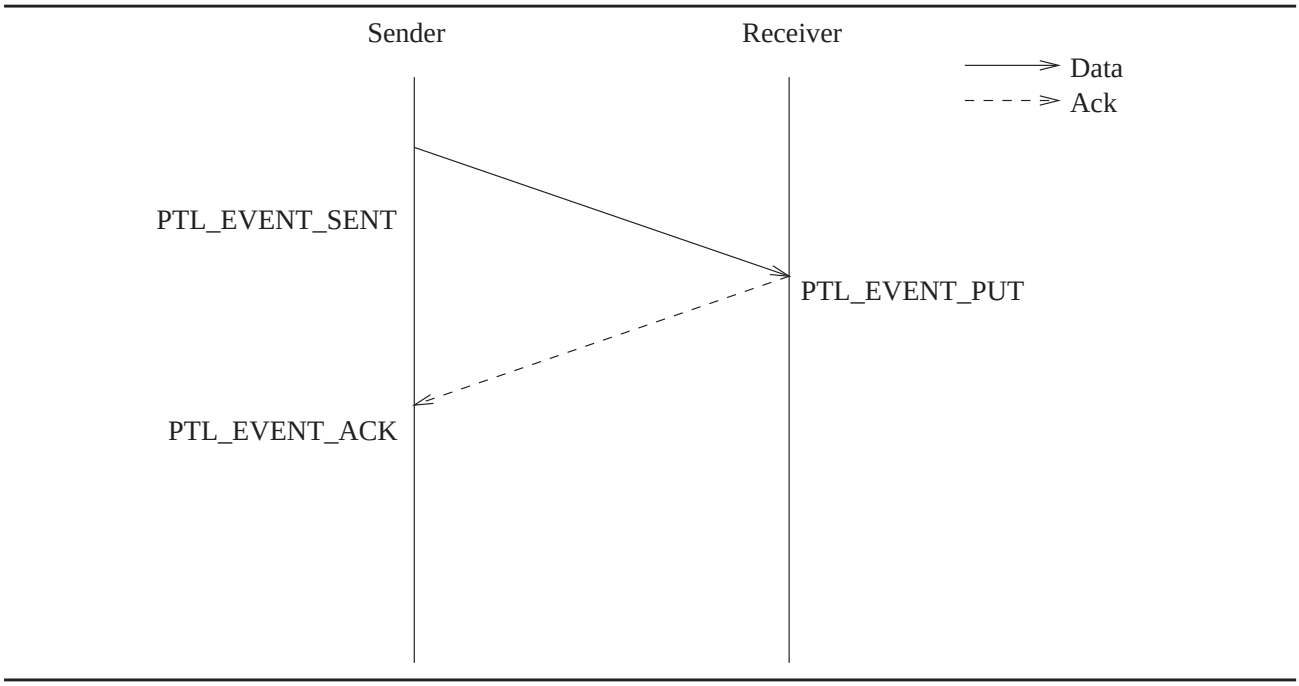

Fig. 6 Long standard mode send (expected).

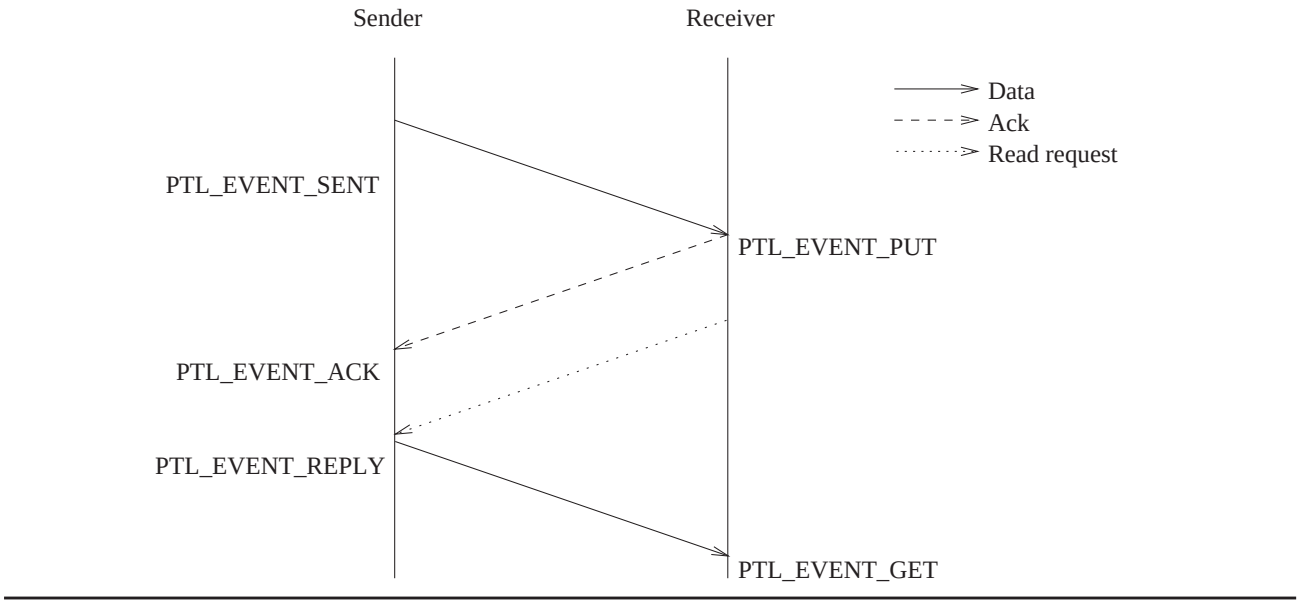

Fig. 7 Long standard mode send (unexpected).

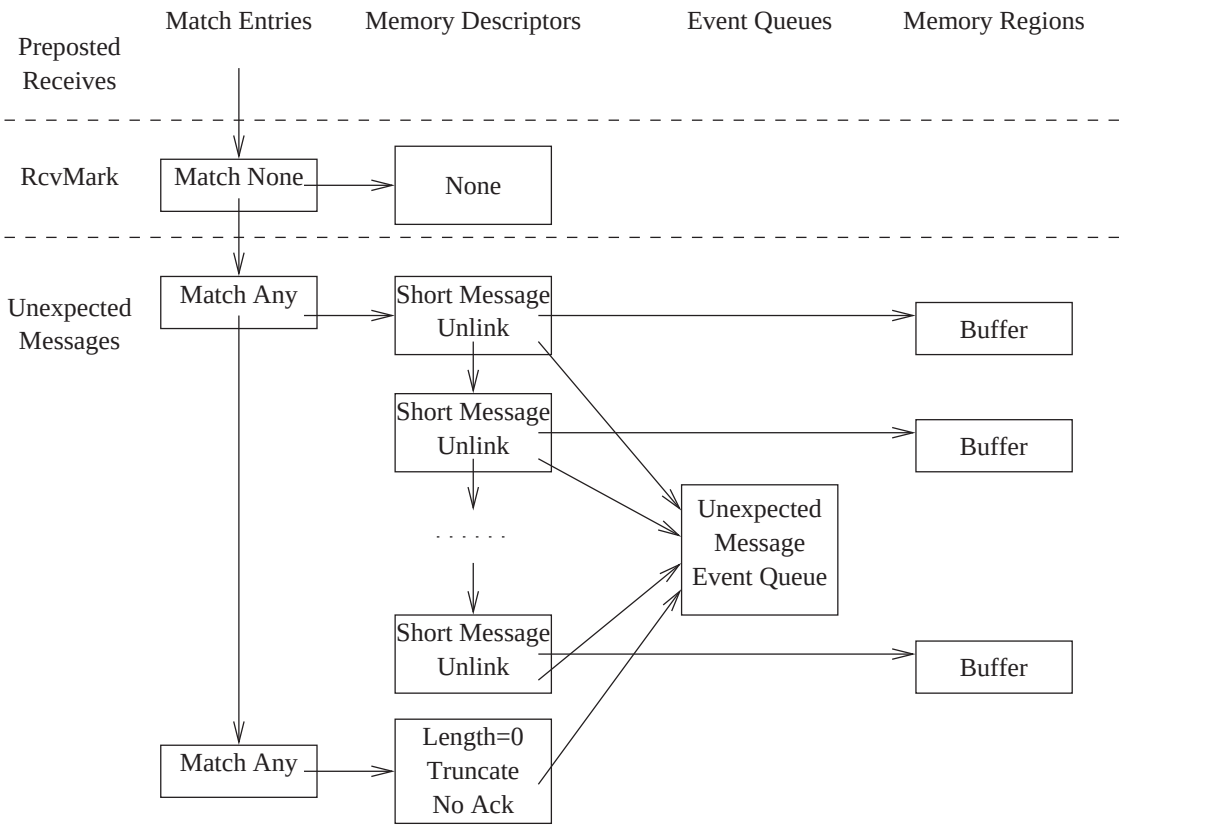

Fig. 8 Message reception in the initial implementation.

by a *mark* entry that separates the two lists, followed by entries for unexpected messages.

When a process calls an MPI receive, we create an MD that describes the user buffer. The MD is configured to accept put operations, generate acknowledgments, unlink when used, and is given an initial threshold of zero (making the MD inactive). The communicator id, source rank within the communicator, and message tag are translated into match bits. We use these match bits along with the MD to build an ME which is inserted just in front of the *mark* entry in the match list.

Next, we search the unexpected messages for a match. In our implementation, unexpected messages can appear in two places. Initially we look for unexpected messages in a queue of Portal events that the MPI library maintains. These events initially appear in an EQ associated with the MDs for unexpected messages. This EQ is shown in Figure 8. Since events are "consumed" when they are removed from the EQ, we must hold on to them if they do not match the receive we are processing. The queue in the MPI library holds on to these unmatched events. If we do not find a matching message in the library queue of events, we check the EQ to see if a matching message has arrived.

We now have an ME and an MD that describe the receive operation to be posted, but we must ensure that no further messages have arrived that match this receive before it can be posted. The process of checking the unexpected queue and posting a receive must be atomic. To accomplish this, Portals has a function that will activate an MD if and only if a given EQ is empty. If there are no pending events, the MD is activated. If there are pending events, the EQ is again checked for a matching message.

If we find a match, we unlink the ME from the match list and take the appropriate action based on the protocol bit in the message. If the message is a long protocol message, we activate the MD and perform a Portal get

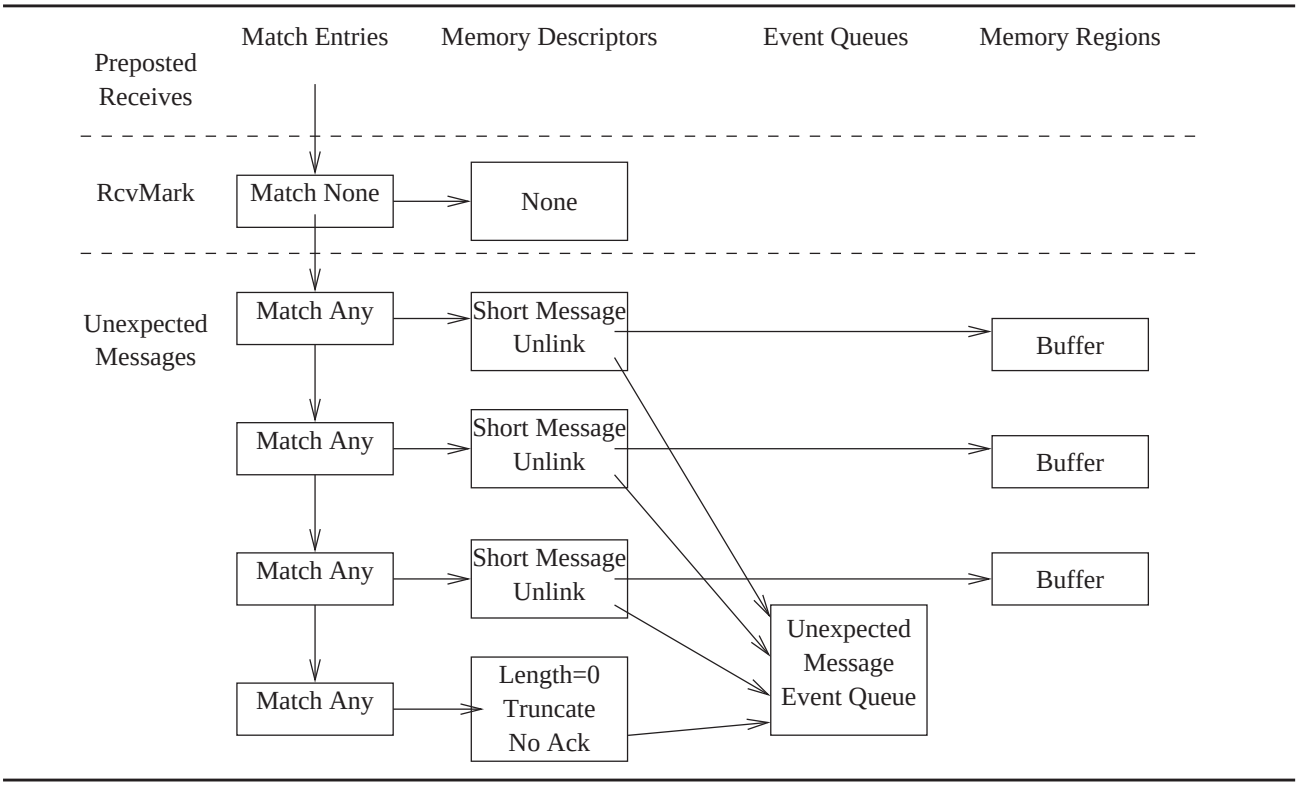

Fig. 9 Message reception in second implementation.

operation, which reads the send buffer from the sender. If the message is a short protocol message, we simply copy the contents of the unexpected buffer into the user's receive buffer. If the header data in the event is nonzero, this indicates that the send is synchronous. In this case, we send a zero-length message to the acknowledgment Portal at the sender using the Portal put operation.

4 Limitations

When the MPI library is initialized, we create two MEs, one for unexpected short protocol messages and another for unexpected long protocol messages. The ME for short messages will match any message that has the short message bit set. There are 1024 MDs attached to this ME. Each MD provides an 8 KB memory buffer, has a threshold of one, is configured to accept put operations, and does not automatically generate an acknowledgment. The match entry for long unexpected messages has an MD of zero bytes with an infinite threshold that is configured to accept and truncate put operations and automatically generate an acknowledgment. All of the MDs for unexpected messages are attached to the same EQ to preserve message ordering. This EQ is created with 2048 entries.

Despite being able to support 1024 outstanding unexpected short messages at any time, this limitation proved to be too restrictive in practice, especially as applications scaled beyond 512 processes. Our application developers typically think of limits in terms of messages size rather then message counts. In our implementation, an unexpected message of 0 bytes would consume an 8 KB MD. To an application developer, this message shouldn't consume any buffer space at all. Moreover, in a NIC-based implementation, 1024 MDs consumes a significant amount of a limited resource, NIC memory, leaving fewer MDs for posted receives.

5 Second Implementation

To address these problems, an additional threshold semantic was added to MDs. An MD could be created with a

maximum offset, or high-water mark. Once this offset was exceeded, the MD would become inactive. Figure 9 illustrates our new strategy for handling short protocol unexpected messages. We now create three MEs for unexpected short messages, all with identical selection criteria. Each ME has an MD attached to it that describes a 2 MB buffer. As unexpected messages come into the MD, they are deposited one after the other until a message causes the maximum offset to be exceeded. When this happens, the MD is unlinked, and the next unexpected short message will fall into the next short unexpected MD. Once all of the unexpected messages have been copied out of the unlinked MD, the ME and MD can be inserted at the end of the short unexpected MEs.

This new strategy allows for the number of unexpected messages to be dependent on space rather than count. It also reduces the number of MDs for short unexpected messages to three, significantly reducing the amount of NIC resources required by MPI. The handling of posted receives and unexpected long protocol messages did not change. This semantic change to Portals eliminated the need for lists of MDs, so the API was changed to allow only a single MD per match entry. In addition, we felt these semantic changes were significant enough to warrant an increase in the version number, so the specification was changed to Portals 3.1.

6 Progress

MPI has asynchronous send and receive calls that allow high quality implementations the opportunity to overlap computation and communication. MPI also defines rules for how asynchronous communication operations should make progress. In particular, progress on outstanding asynchronous communication operations is independent of calls into the MPI library.

Every OS-bypass MPI implementation described in the current Literature (Lauria and Chien, 1997; O'Carroll et al, 1998; Prylli et al., 1999; Dimitrov and Skjellum, 1999) requires application processing to move data. These implementations typically use a two-level protocol, where short messages are sent eagerly and long messages are sent using a rendezvous protocol. As in our implementation, short eager messages are buffered at the receiver and copied by the application into the appropriate receive buffer after context and tag matching occur.

In the long message rendezvous protocol, the sender sends a request to the receiver. This request is recognized by the application, the context and tag matching occur, and when the appropriate receive buffer is found, a message is sent back to sender, indicating the exact location in memory where the data can be delivered. However, because the application must be involved in these transfers, the opportunity for significant overlap is lost.

Since our implementation sends long messages eagerly, it can overlap the data transfer at the receiver, provided the message is expected. If the message is unexpected, it can overlap the transfer at the sender by using a get operation. One could argue that this approach makes progress at the expense of wasting network resources. The possibility to consume network bandwidth and create contention is greater than if a rendezvous protocol were used. However, since Portals is based on expected messages, we optimize for the case when MPI receives are pre-posted, and encourage our application developers to follow this practice. We are currently investigating the effect of using an eager protocol for long messages on application benchmarks and real applications. We expect that the benefits of overlapping computation and communication will outweigh such drawbacks as increased network contention.

7 Performance

In this section, we provide some initial performance results from an experimental implementation of Portals 3.0 on Myrinet. These results demonstrate the ability to perform Portals processing on a programmable NIC.

Each node used for gathering these performance results contained a 617 MHz Alpha EV67 processor with 256 MB of main memory and Myrinet LANai 9 NIC. Nodes were connected using a 64-port Mesh64 switch.

For this implementation of Portals on Myrinet, all reliability and flow control is performed on the card via a Myrinet Control Program (MCP). Processing of Portals messages can occur either in the MCP or via a Linux kernel module. A process can choose to have all processing of Portals messages occur on the card, which is expected to incur minimal host processor overhead, or have intitial processing done in an interrupt handler on the host. Once Portals processing has occurred in the interrupt handler, data is transferred directly from the network into user space. Both of these methods of processing a Portals message employ OS-bypass, since the OS is not involved in the transfer of data once the final destination is determined. This is an experimental implementation of Portals that is currently running on a small development system. It is currently limited to being able to send and receive messages into a 4 MB physically contiguous region of memory.

Figure 10 shows latency performance from a traditional ping-pong benchmark. The graph shows the half round-trip latency performance of the Portals MCP implementation where Portals processing occurs in the MCP and in the kernel, as well as the performance of our MPI implementation using each of these two strategies. The

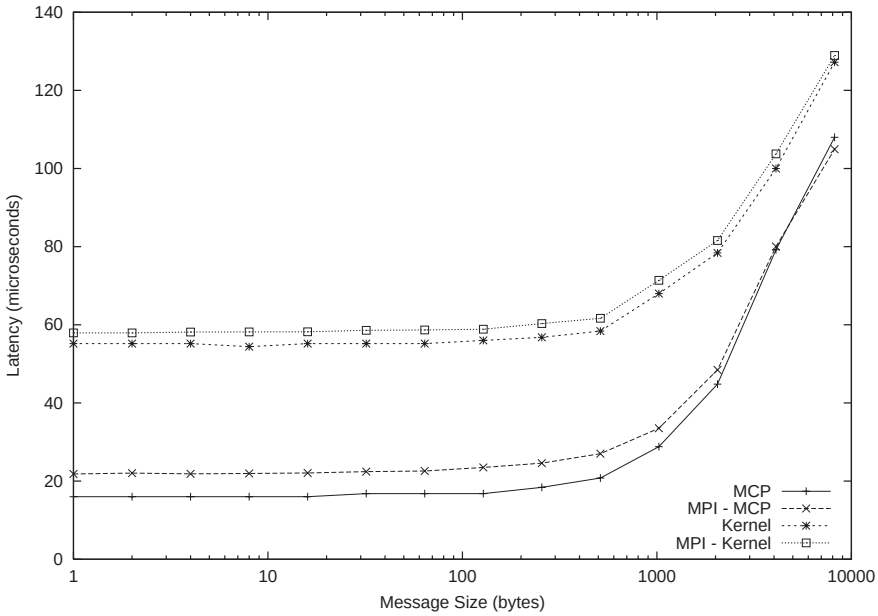

Fig. 10 MPI half round-trip latency performance.

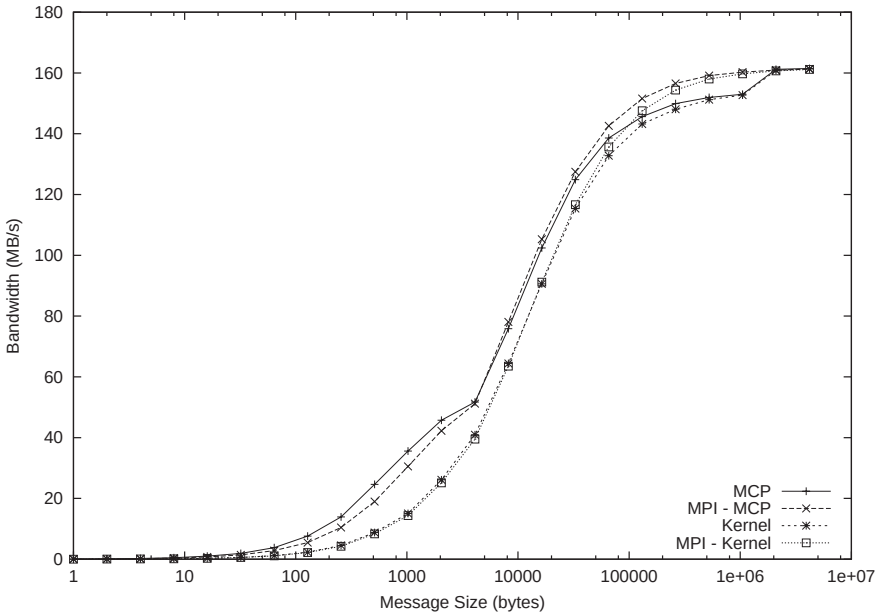

Fig. 11 MPI bandwith performance.

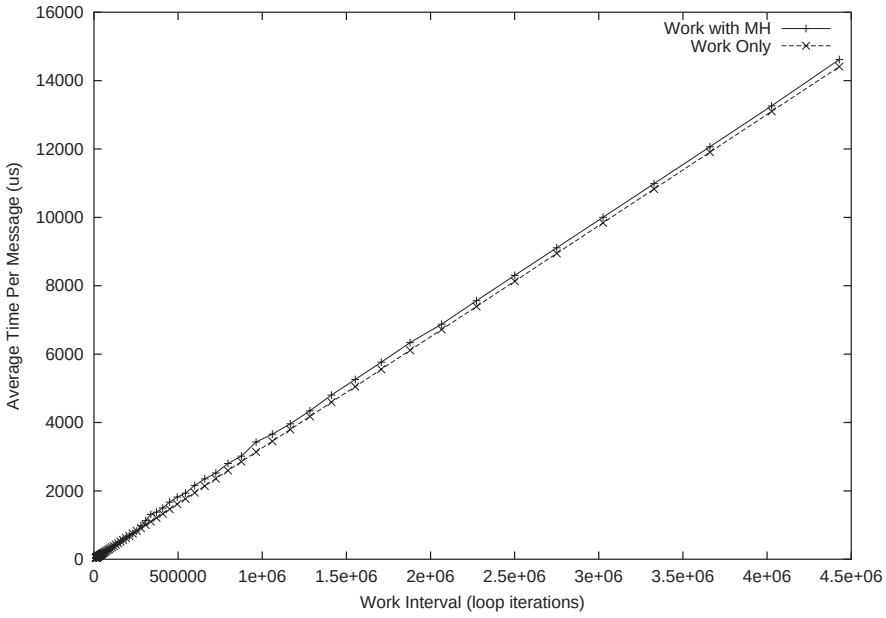

Fig. 12 Host CPU overhead (host).

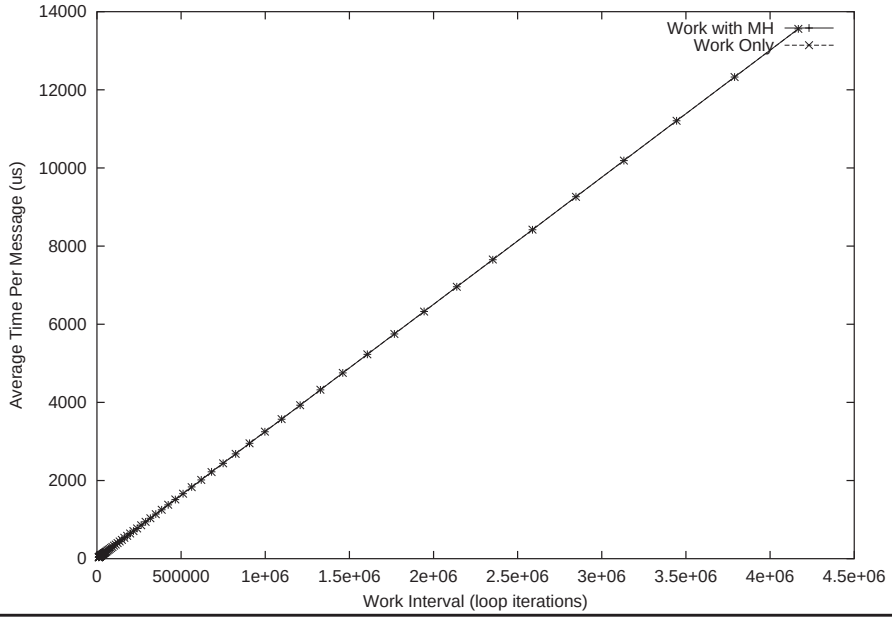

Fig. 13 Host CPU overhead (NIC).

zero-length Portals latency for the case where Portals processing occurs in the MCP is 17 μsec, and the overhead of the MPI implementation adds approximately 5 μsec. Figure 11 shows the corresponding bandwidth performance.

The results in Figures 12 and 13 are from the COMB benchmark suite (Lawry et al., 2002), which characterizes the amount of overlap of computation and communication that an MPI implementation can support. These graphs show the time to complete a given work interval with and without communication. For this data, 100 KB messages were being received while performing work. The difference between the two times is due to overhead incurred on the host CPU. Figure 12 shows the overhead for the case where Portals processing is performed in the kernel. The impact of the interrupt handler is clearly evident. Figure 13 shows the overhead for Portals processing in the MCP. In this case, there is almost no impact of message passing on the host CPU.

8 Summary

This paper has described implementations of MPI on the Portals data movement layer. We have illustrated how Portals provides the necessary building blocks and associated semantics to implement MPI very efficiently. In particular, these building blocks can be implemented on intelligent network interfaces to provide the necessary protocol processing for MPI without being specific to MPI. The implementations described in this paper have been in production use on a 1792-node Alpha/Myrinet Linux cluster at Sandia National Laboratories for more than two years. We have also shown preliminary performance data from an implementation of Portals for Myrinet that demonstrates the ability to efficiently perform Portals address processing on a programmable network interface card.

9 Ongoing and Future Work

Although the Portals interface was originally intended to be a user-level data movement layer, the Lustre project has adopted Portals as the underlying interface for its distributed file system (Braam et al., 2002). For some systems, this requires the Portals interface to be used at the kernel level rather than at the user level. Supporting Portals in-kernel has had a slight impact on the interface and semantics. For example, the 3.0 and 3.1 versions had semantics that implied direct integration with a runtime system. Processes could be addressed based on their rank within a parallel job. Because these ties do not make sense when Portals is used at the kernel level, we eliminated most of the places where the Portals interface had direct ties to a runtime system. This change had minimal impact on the semantics, but we subsequently updated the version number to 3.2 to reflect the enhancement. The Lustre project is now working on various kernel- and user-level implementations of Portals in order to support their file system on various network and storage hardware.

Cray, Inc. has also adopted Portals as the high-performance network layer for Sandia's ASCI Red Storm machine, which is to be delivered in 2004. Cray and Sandia are currently working on an implementation of Portals 3.2 for a custom network interface that utilizes the HyperTransport links on the AMD Opteron processor.

Sandia and the University of New Mexico are also working on implementations of Portals for various commodity and specialized networks, including gigabit Ethernet and Quadrics (Petrini et al., 2002). There is also an effort underway at Los Alamos National Laboratory to implement Portals 3.2 on Infiniband (2000) hardware.

Sandia is also working on various software layers that utilize the Portals interface. We are continuing to develop the Cplant™ parallel runtime system, and are working on supporting other higher-level message passing and one-sided data movement layers. MPI Software Technology, Inc. currently has an implementation of MPI 1.2 and supports portions of MPI 2.0, including one-sided communications, on top of Portals. We are also working to support the next generation MPI implementation, MPICH-2 (Gropp, 2002), on top of Portals.

The current version of the Portals specification and a reference implementation that supports several different networks are available under the GNU LGPL from http://sourceforge.net/projects/sandiaportals.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the contribution of Mike Levenhagen and Trammell Hudson to Portals and this research. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

BIOGRAPHIES

Ron Brightwell is a Principal Member of Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico. He received the BS degree in mathematics in 1991 and the MS degree in computer science in 1994, both from Mississippi State University. Since joining Sandia in 1995, he has participated in several research and development projects related to system software for scientific parallel computing systems. He has designed and developed high-performance implementations of the Message Passing Interface (MPI) standard

on several large-scale, massively parallel computing platforms, including the Cray T3D and T3E, the Intel Paragon and TeraFLOPS (ASCI/Red), and Sandia's Computational Plant clusters. His research interests include high-performance, scalable communication interfaces and protocols for system area networks, operating systems for massively parallel processing machines, global address space programming models, and parallel program performance analysis libraries and tools. He is also currently enrolled in the Ph.D. program at the University of New Mexico.

Rolf Riesen is a Principal Member of Technical Staff at Sandia National Laboratories in Albuquerque, New Mexico. He holds a Ph.D. degree in computer science from the University of New Mexico (UNM). His research interests include message passing systems, operating systems, and runtime software for massively parallel computers. Over the last ten years in this field he has primarily concentrated on topics related to efficient, scalable message-passing and interactions at the software/hardware boundary. Dr. Riesen has been a key member of the design team for SUNMOS (Sandia/UNM OS) for the nCUBE 2 and the Intel Paragon, as well as Puma/Cougar, the second generation light weight kernel for the Intel ASCI Red machine. He is also a key designer of the Portals message passing mechanism. He has been a principal designer of Computational plant, the largest commodity cluster for scientific applications in the world. He was involved in the overall design and the low level message passing layers, including the scalable wire protocol used by Computational plant.

Arthur B. (Barney) Maccabe is an Associate Professor in the Computer Science Department at the University of New Mexico. His primary research interests are in the areas of operating systems and the design of communication protocols with an focus on high-performance computing. He has significant experience in the design and implementation of lightweight kernels for massively parallel systems. Professor Maccabe was a principle architect of SUNMOS (Sandia/UNM OS) for the Intel Paragon and Puma/Cougar for the Intel Teraflop (ASCI/Red) at Sandia National Laboratories. He was also a member of the original MPI forum. In recent years, he has been involved in the development of the Portals API and the implementation high-performance communication protocols.

REFERENCES

Boden, N., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N., and Su, W. 1995. Myrinet – A gigabit-per-second local-area network. *IEEE Micro*, 15(1): 29–36.

Braam, P., Brightwell, R.B., and Schwan, P. 2002. Portals and networking for the Lustre file system. In: *Proceedings of IEEE International Conference on Cluster Computing*.

Brightwell, R.B. and Fisk, L.A. 2001. Scalable parallel application launch on Cplant™. In: *Proceedings of the SC2001 Conference on High Performance Networking and Computing*, ACM Press and IEEE Computer Society Press.

Brightwell, R.B., Fisk, L.A., Greenberg, D.S., Hudson, T.B., Levenhagen, M.J., Maccabe, A.B., and Riesen, R.E. 2000. Massively parallel computing using commodity components. *Parallel Computing*, 26(2–3): 243–266.

Brightwell, R.B., Hudson, T.B., Maccabe, A.B., and Riesen, R.E. 1999. *The Portals 3.0 Message Passing Interface*, Technical Report SAND99-2959, Sandia National Laboratories.

Brightwell, R.B., Lawry, W., Maccabe, A.B., and Riesen, R.E. 2002. Portals 3.0: Protocol building blocks for low overhead communication. In: *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, Fort Lauderdale, FL.

Compaq, Microsoft, and Intel. 1997. *Virtual Interface Architecture Specification Version 1.0*.

Dimitrov, R. and Skjellum, A. 1999. An efficient MPI implementation for virtual interface (VI) architecture-enabled cluster computing. In: *Proceedings of the Third MPI Developers' and Users' Conference*, pp. 15–24.

Gropp, W. 2002. MPICH2: A new start for MPI implementations. In: *Lecture Notes In Computer Science, 2474: Recent Advances in Parallel Virtual Machine and Message Passing Interface*, edited by D. Kranzlmüller, P. Kacsuk, J. Dongarra, and J. Volkert., p. 7, Springer-Verlag.

Gropp, W., Lusk, E., Doss, N., and Skjellum, A. 1996. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828.

InfiniBand Trade Association. 2000. InfiniBand Architecture Specification, Version 1.0. (http://www.infinibandta.org/).

Ishikawa, Y., Tezuka, H., and Hori, A. 1996. *PM: A high-performance communication library for multi-user parallel envrionments*, Technical Report TR-96015, Real World Computing Partnership.

Lauria, M. and Chien, A.A. 1997. MPI-FM: High performance MPI on workstation clusters. *Journal of Parallel and Distributed Computing*, 40(1): 4–18.

Lawry, W., Wilson, C., Maccabe, A.B., and Brightwell, R.B. 2002. COMB: A portable benchmark suite for assessing MPI overlap. In: *Proceedings of IEEE International Conference on Cluster Computing*.

Maccabe, A.B., McCurley, K.S., Riesen, R.E., and Wheat, S.R. 1994. SUNMOS for the Intel Paragon: A brief user's guide. In: *Proceedings of the Intel Supercomputer Users' Group Annual North America Users' Conference*, pp. 245–251.

Message Passing Interface Forum. 1994. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8: (3–4): 159–416

Myricom, Inc. 1997. *The GM Message Passing System*.

O'Carroll, F., Hori, A., Tezuka, H., and Ishikawa, Y. 1998. The design and implementation of zero copy MPI using commodity hardware with a high performance network. In: *Proceedings of the 1998 International Conference on Supercomputing*, Melbourne, Australia, pp. 243–250.

Pakin, S., Karamcheti, V., and Chien, A.A. 1997. Fast messages (FM): Efficient, portable communication for workstation clusters and massively parallel processors. *IEEE Concurrency*, 40(1): 4–18.

Petrini, F., Chun Feng, Wu, Hoisie, A., Coll, S., and Frachtenberg, E. 2002. The Quadrics Network: High-performance clustering technology. *IEEE Micro*, 22(1): 46–57.

Prylli, L. and Tourancheau, B. 1998. BIP: A new protocol designed for high performance networking on Myrinet. *Lecture Notes in Computer Science*, 1388: 472.

Prylli, L., Tourancheau, B., and Westrelin, R. 1999. The design for a high performance MPI implementation on the Myrinet Network. In: *Proceedings of the 6th European PVM/MPI Users' Group*, pp. 223–230.

Shuler, P.L., Jong, C., Riesen, R.E., van Dresser, D., Maccabe, A.B., Fisk, L.A., and Stallcup, T.M. 1995. The Puma operating system for massively parallel computers. In: *Proceedings of the 1995 Intel Supercomputer User's Group Conference*.

von Eicken, T., Basu, A., Buch, V., and Vogels, W,. 1995. U-Net: A user-level network interface for parallel and distributed computing. In: *Proceedings of the Fifteenth Symposium on Operating System Principles*, pp. 40–53.

von Eicken, T., Culler, D.E., Goldstein, S.C., and Schauser, K.E. 1992. Active messages: A mechanism for integrated communications and computation. In: *Proceedings of the 19th International Symposium on Computer Architecture*, vol. 0, pp. 256–266.